Common Mechanisms for Supporting Fault Tolerance in DSM and Message Passing Systems

Ramamurthy Badrinath, Christine Morin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Common Mechanisms for Supporting Fault Tolerance in DSM and Message Passing Systems

Ramamurthy Badrinath and Christine Morin

N˚4613

Novembre 2002

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4613--FR+ENG

Common Mechanisms for Supporting Fault Tolerance in DSM and Message Passing Systems

Ramamurthy Badrinath * and Christine Morin†

Thème 1 — Réseaux et systèmes
Projet PARIS

Rapport de recherche n ° 4613 — Novembre 2002 — 15 pages

Abstract: Backward error recovery involving checkpointing and restart of tasks is an important component of any system providing fault tolerance to applications distributed over a network. A central problem to checkpointing and recovery is the ability to track dependencies and arrive at a consistent global checkpoint. Traditionally literature treats one of either distributed shared memory (DSM) or message passing as the interprocess communication mechanism when considering the issue of fault tolerance. This paper describes preliminary investigation into common mechanisms that can be implemented to support a wide variety of protocols in both shared memory and message passing systems. In effect it can be used in a system that combines both these IPC mechanisms.

Key-words: *Distributed systems, Fault tolerance, Checkpointing, Rollback recovery, Dependency tracking, Message passing, Shared memory*

(Résumé : tsvp)

* The author is currently on leave from the CSE Department, Indian Institute of Technology, Kharagpur - 721302, INDIA
† IRISA/INRIA {Ramamurthy.Badrinath, Christine.Morin}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Mécanismes de tolérance aux fautes communs aux systèmes à mémoire partagée répartie et aux systèmes communiquant par messages

Résumé : *Le recouvrement arrière fondé sur la sauvegarde de points de reprise à partir desquels les tâches peuvent être redémarrées constitue un composant important d'un système offrant de la tolérance aux fautes à des applications distribuées. Un problème central dans les techniques de sauvegarde et de restauration de points de reprise est la capacité à détecter les dépendances entre processus communicants pour construire un point de reprise global cohérent. Traditionnellement, la littérature traite séparément les systèmes fondés sur une mémoire partagée répartie et ceux fondés sur l'échange de messages pour les communications inter-processus lorsqu'il s'agit d'étudier ces mécanismes de tolérance aux fautes. Dans ce rapport, nous présentons nos premières investigations dans la conception de mécanismes communs aux systèmes communiquant par mémoire partagée et à ceux communiquant par messages pour le support d'une gamme étendue de protocoles de recouvrement arrière. Ainsi, il est possible d'utiliser ces mécanismes dans un système qui combine ces deux schémas de communication inter-processus.*

Mots-clé : *Systèmes distribués, Tolérance aux fautes, Recouvrement arrière, Point de reprise, Détection de dépendances, Communication par messages, Mémoire partagée*

1 Introduction

In distributed systems, the concept of a consistent global state is an important component in algorithms that recover the system from a faulty state. A consistent global state[3] is a set of local states which maintain causal consistency. Causal interrelation between states of different tasks arises from inter-process communication (IPC).

One encounters two kinds of distributed systems depending on the type of IPC used - one which uses message passing (called **MP** systems in this paper), and one where there is a distributed shared virtual memory for all tasks in the system (called **DSM** systems in this paper). For this study we consider shared memory reads and writes, and explicit message passing using send-receive pairs as the only form of IPC. We refer to these collectively as *interactions*.

A *local state* consists of the processor state and the private memory state of a task. In pure MP systems, all memory of a task is private. In case of DSM systems private memory for a task is usually only its stack.

A *local checkpoint* is a saved local state. For MP systems the set of local checkpoints which are consistent forms a global state to which the system may recover. In the case of DSM systems we must additionally restore the shared virtual memory to a state that is consistent with the states of all the tasks. In fact one can view the units of shared memory as entities themselves like tasks and make sure these are consistent with the tasks' local states on recovery. This is the view we take in this paper.

The objective of checkpointing and recovery algorithms is to compute the *latest* global state to which the system can recover on failure. This is called the *recovery line*. It is not our purpose in this paper to describe or compare different checkpointing or recovery protocols. These are available in excellent surveys for both MP [4] and DSM [9] systems. Instead we focus on describing mechanisms that can be used in building support for these protocols. Still for completeness sake let us describe the major classification of algorithms in this area. Broadly speaking there three classes of checkpointing (and corresponding recovery) protocols for tasks:

- *Coordinated checkpointing:* Here tasks coordinate with each other so that at any time the set of latest checkpoints for all tasks forms a recovery line. The advantage of this mechanism is that recovery is simple and it

keeps only the latest checkpoint of each task. The disadvantage is the fact that coordination is needed to establish such a checkpoint.

- *Uncoordinated checkpointing:* Here tasks take checkpoints independently. The advantage is that checkpointing involves no coordination and hence is simpler. The disadvantage is that recovery is complex. Usually recovery requires us to maintain a number of checkpoints for each task as well as a history of the interactions.

- *Communication Induced Checkpointing:* Here tasks take independent checkpoints, but in addition to these they also take additional checkpoints referred to as *forced* checkpoints. Thus one may view this as introducing some amount of coordination into an uncoordinated checkpointing protocol. The forced checkpoints help make the recovery line progress. Hence we may not need to maintain as much history information as in the case of uncoordinated checkpointing.

In all the above cases dependency tracking plays an important role in the checkpointing and recovery protocols and in optimizations of them for implementation.

In our discussion we will be considering two kinds of entities that need checkpointing - tasks (with private states), and shared memory pages. Though choosing pages as the unit of memory can lead to false sharing, we prefer to use this model because it is easily supportable in practice. Each entity (page and task) will maintain enough information to know what other entities it *directly* depends on. This information will be used then to compute globally consistent states required for recovery.

In this paper we propose mechanisms that can be used to support a wide variety of checkpointing and recovery protocols on both DSM and MP systems. The rest of the paper is organized as follows. In Section 2 we present our communication and state model and introduce the formalism to represent dependency. Section 3 discusses the proposed mechanism and Section 4 shows its application to the major classes of protocols. We then discuss implementation issues in Section 5. Finally, in Section 6 we compare the work in this paper to other related work.

2 Communication and State Model

We now describe a model to clarify our notion of entity state and of IPC. We can then relate our mechanisms to this model and describe how it works in practice.

We consider direct communication between tasks as consisting of a *message* send-receive pair. The communication channel is considered unreliable, implying that messages may be lost. Each message receive generates a dependency. We say that there is a dependency from the sender to the receiver, or that the receiver is dependent on the sender. If the sender is rolled back to a state before the send, then the receiver needs to be rolled back to a state before the receive. This maintains meaningful causal relationship.

Direct communication also happens between a shared memory page and a task, caused by read and write operations. On a *read* from a page a task state becomes dependent on the corresponding page state. If the page is rolled back to a state prior to the read then the corresponding task may also need to be rolled back to a state prior to its read. This is very similar to the send-receive pair described above, with the additional note that there is no equivalent of a "lost" message. Note that not every interaction results in a new dependency. This is an implementation issue that is discussed in Sections 3 and 5.

A *write* to a page presents an interesting difference from message passing. A write always succeeds and when a write happens, it causes a *two way dependence* as noted by Gunaseelan and LeBlanc[5]. This means that after the write, if the task is rolled back to a state before the write, then so must the page. Similarly if a page is rolled back to a state before the write, then so must the task.

These three varieties of dependencies are depicted in Figure 1. The figure also shows the other event of interest to us which is the checkpoint event. The checkpoints of a tasks are serially ordered with $c_{i,j}$ representing the j^{th} checkpoint of entity i. We discuss the example in more detail after introducing some formalism.

We can now formalize the above dependencies in the context of checkpointing and rollback recovery. We do so by extending the now well established "happened before" relation [1, 11]. We note that when the system recovers from a failure each entity is either left untouched in its current state or is rolled back

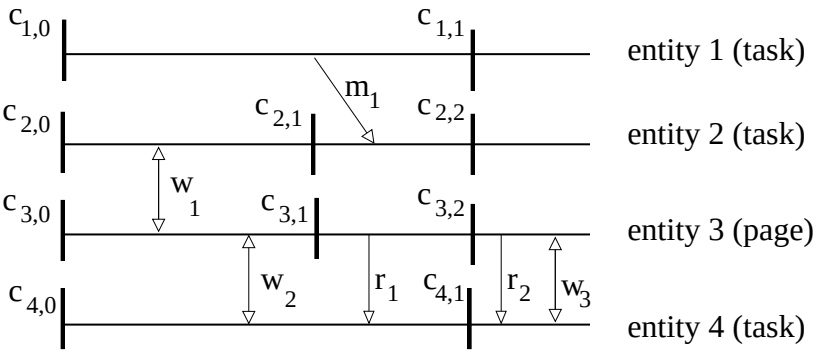

Figure 1: An illustration of interactions causing interdependencies. $c_{i,j}$ is the j^{th} checkpoint of entity i. w_1, w_2, w_3 are shared memory writes, r_1, r_2 are shared memory reads and m_1 is a message passed from a sender to a receiver.

to a checkpoint. We may assume for convenience of argument that the current state of an unfailed entity is a temporary checkpoint too, and hence we will focus on a formalism that establishes dependencies between checkpoints.

We say that $c_{l,m}$ *depends on* $c_{i,j}$, denoted $c_{i,j} \rightarrow c_{l,m}$ if and only if one of the following conditions holds:

1. $i = l$ and $j < m$
2. $i \neq l$ and entity l and i are tasks and task i sends a message after $c_{i,j}$ that task l receives before $c_{l,m}$.
3. $i \neq l$ and entity i is a page and entity l is a task and the task reads the page after the page's checkpoint $c_{i,j}$ but before the task's checkpoint $c_{l,m}$.
4. $i \neq l$ and one of the entities is a page and the other is a task, and the task writes to the page after $c_{i,j}$ and before $c_{l,m}$.

Conditions 1 and 2 are typically used to define dependencies in message passing systems. Conditions 3 and 4 follow from our description of dependence between tasks and shared memory. Condition 4 brings out the property that a write creates a two way dependency. Referring to Figure 1, $c_{1,0} \rightarrow c_{1,1}$ by condition 1, $c_{1,0} \rightarrow c_{2,2}$ by condition 2 (caused by m_1), $c_{3,1} \rightarrow c_{4,1}$ by condition

3 (caused by r_1), and $c_{2,0} \rightarrow c_{3,1}$ by condition 4 (caused by w_1). Also note that the other effect of w_1 is that $c_{3,0} \rightarrow c_{2,1}$.

An interesting result of viewing the page as a separate entity is that certain "Write after Write" dependencies that arise between tasks [2] are naturally handled by this model. In the example of Figure 1, if entity 4 fails immediately after w_2, and rolls back to $c_{4,0}$ then [2] tells us that entity 2 must recover to the state prior to w_1, i.e., to $c_{2,0}$. This is explicit in our model. On entity 4 failing, we must rollback entity 3 to $c_{3,0}$ which in turn implies that we must rollback entity 2 to $c_{2,0}$. In fact our model also tells us that if after w_1 and w_2 if entity 2 fails before $c_{2,1}$ then we must roll back entities 3 and 4 to $c_{3,0}$ and $c_{4,0}$ respectively.

For this paper we assume a *fail stop* failure model. Our mechanisms do not assume any specific limit on the number of failures itself. That is an issue that will depend on the details of the checkpoint and recovery protocols.

3 Mechanism for Dependency Tracking for Recovery Line Computation

In this section we describe the dependency tracking mechanism and describe how it can be used to compute the recovery line for both MP and DSM systems. Our mechanisms are based on ideas from the work of Baldoni *et. al.* [1], and that of Gunaseelan and LeBlanc [5].

3.1 Basic Mechanism

We associate with each entity (i.e., each page and each task) an integer called a *sequence number*. This is initialized to 1 when the entity begins its existence. This is incremented only when the entity takes a checkpoint. The sequence number of the source of a dependency is delivered to the destination of the dependency whenever a dependency is created. For instance in Figure 1, r_1 causes the current sequence number of entity 3 to be delivered to entity 4. A write will cause sequence numbers of the task to be made available to the page and *vice-versa*. Over time an entity receives a number of such sequence numbers from other entities. These are stored in a vector called the *direct*

dependence vector (**DDV**) of the entity. If there is a dependence from entity i to j (for instance a message is sent from task i to task j), and the sequence number (of i) sent in the interaction is sn, and $ddv[j]$ is the local dependence vector with entity j, then on the event at j, we execute the following code:

$$ddv[j][i] = \max\{ddv[j][i], sn\}$$

Initially for an entity j, $ddv[j]$ contains all zeros. One may consider the initial image of an entity to be its zero-th checkpoint just for ease of argument. The local value of sn will always be the *checkpoint number* of the next checkpoint to be taken for the entity. Whenever we checkpoint an entity j we store along with the local checkpoint, the corresponding DDV, i.e., $ddv[j]$. This saved DDV is called the *timestamp* of the corresponding checkpoint. Whenever a checkpoint is taken at a node j we execute the following code:

$$ddv[j][j] = sn; sn++; \texttt{save the timestamp } ddv[j].$$

We will refer to the timestamp associated with the k^{th} checkpoint of entity j as $ddv_k[j]$. Note that $ddv_k[j][j] == k$. For completeness sake, the timestamp for the zero-th checkpoint for all entities has all zeros.

This mechanism essentially records direct dependencies between checkpoints of various entities. So for instance if entity i decides to rollback to checkpoint number n, then for entity j if $ddv[j][i] > n$, then clearly entity j needs to rollback to a checkpointed state for which $ddv[j][i] \leq n$. Of course this covers only direct dependencies. The recovery line computation is the responsibility of the recovery protocol.

With this mechanism in place it is possible to detect all direct dependencies. Yet, in the case of shared memory, some optimization is possible. Consider a page that has not been changed between the two checkpoints $c_{i,j}$ and $c_{i,k}(k > j)$ for the page, then clearly any read after $c_{i,k}$ still refers to the version checkpointed at $c_{i,j}$ provided the page has not yet been modified before the read after $c_{i,k}$. Recording the newer sequence number by the reader results in an artificial dependence. Hence in a read of a page we may prefer to use an older value than the actual sequence number of the page. For this we introduce for each page another counter called the *last write number* (**lwn**). On a read the *lwn* of the page is 'received' by the reading task rather than the *sn* of the page. Also this means that on write to a page, the page must update *lwn* to its current value of *sn*. Typically, algorithms such as that of Janakiraman and Tamir [6] use a bit called the *dirty since checkpoint* (or **dsc**) bit to optimize

the overhead of checkpointing. This bit is set to zero when a checkpoint is taken and set to one when a write is done. One may note that the value of the *dsc* bit is simply the value of the boolean expression $lwn == sn$. Note that not every interaction incurs the overhead mentioned in this section. This is an implementation issue that we discuss further in Section 5.

3.2 Usage in Checkpointing and Dependency Tracking

A local checkpoint for a task is its current private state and the corresponding DDV. A local checkpoint for a page is the current page content, the DDV and the *lwn*. Notice that the value of *sn* is in the DDV itself.

With this we are now in a position to track dependencies in the system which could be caused by message passing or shared memory or a combination of both. Thanks to the result by Wang *et. al.*[11], the direct dependency information is sufficient to detect the recovery line. In the next section we will show how this may be used in a few typical scenarios.

4 Usages in Different Protocol Families

It is possible to use these mechanisms to implement several well established protocols and their optimizations for recovery. We take the three families of protocols - coordinated checkpointing, communication induced checkpointing and uncoordinated checkpointing and corresponding recovery strategies.

4.1 Coordinated Checkpointing

In the case of coordinated checkpointing whenever the system makes a decision to checkpoint one or more entities, it will make sure that the resulting set of checkpoints (one from each entity) forms a recovery line. This requires coordination among potentially all tasks during the checkpointing phase. The two common optimizations that are necessary for this to be efficient in practice are (1) ability to detect which entities need to be checkpointed, and (2) ability to save the checkpoints asynchronously to the storage. The current work is relevant to the former of these two points. If an entity i takes a checkpoint with the corresponding timestamp $ddv_k[i]$, then for each other entity j if

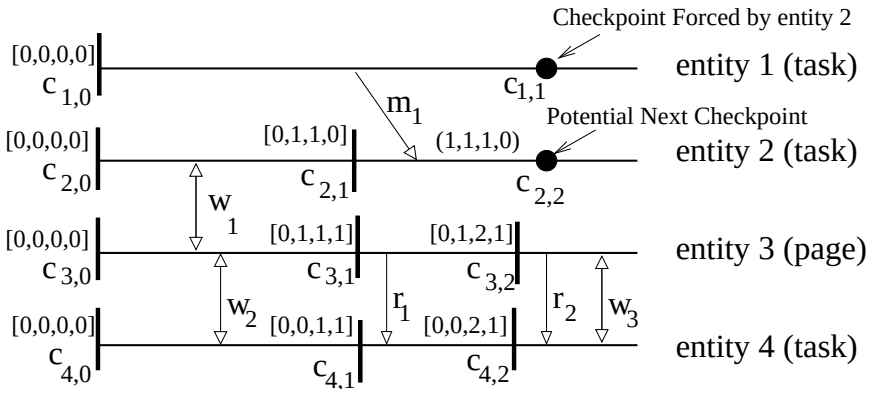

Figure 2: An example with a possible sequence of events in a coordinated checkpointing scenario. Corresponding to each checkpoint the timestamp is shown above the checkpoint label, in square brackets. The DDV for entity 2 immediately after the receipt of m_1 is shown in round brackets.

$ddv_{k-1}[i][j] \neq ddv_k[i][j]$, then entity j must be checkpointed as well, and this protocol has to be applied recursively at entity j. The entities corresponding to the tree of dependencies thus generated can be coordinated and we can generate a coordinated checkpoint. If the entity is a page for which $lwn == sn$, then it means that no updating of the page content information is needed[6].

Consider our example in Figure 2. If the latest checkpoints are $\{c_{1,0}, c_{2,1}, c_{3,2}, c_{4,2}\}$, then if entity 2 decides to take the checkpoint $c_{2,2}$ as indicated, then it forces entity 1 to take the checkpoint $c_{1,1}$ because $ddv[2][1] = 1$ but $ddv_1[2][1] = 0$.

For recovery the solution is quite straight forward. The fault detector or some other recovery management mechanism will detect identities of the faulted entities and broadcast them to all unfailed entities. If a non faulty entity i receives notification that entity j is rolling back, then if $ddv[i][j] \neq ddv_{sn-1}[i][j]$, where sn is the current sequence number at i, then clearly entity i has received a dependency from entity j since its last checkpoint, and hence entity i has to rollback as well. This needs to be done recursively just as in the checkpointing phase.

Consider our example in figure 3. If the set of latest checkpoints is $\{c_{1,1}, c_{2,2}, c_{3,2}, c_{4,2}\}$, then if entity 3 failed after the write w_3 then the recovery would require entity 3 to restart at checkpointed state $c_{3,2}$. When this information reaches entity 4,

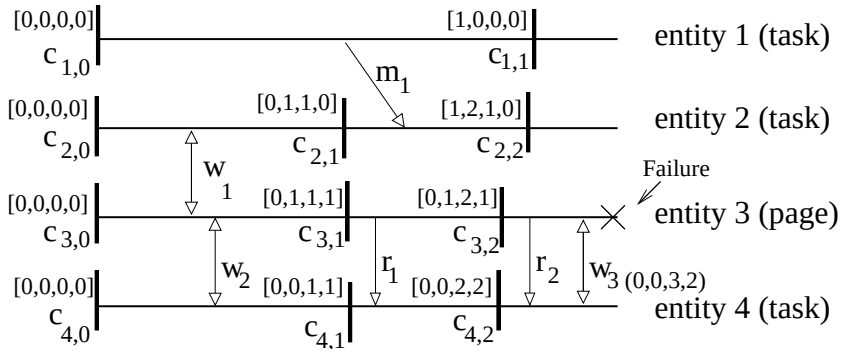

Figure 3: A recovery scenario for coordinated checkpointing. The vector in the round brackets is the DDV at entity 4 after w_3.

it compares locally the value of $ddv[4][3]$ and $ddv_2[4][3]$ and find them to be 3 and 2 respectively, requiring it to rollback to its previous checkpoint $c_{4,2}$. On the contrary, note that if the failure was after r_2 but before w_3, then entity 4 would have only received the lwn value with the page read which means $ddv[4][3]$ would only be 2, hence not requiring any action on the part of entity 4.

4.2 Communication Induced Checkpointing

The checkpoints in the case of communication induced checkpointing are of two kinds - independent and forced. When an entity independently decides to checkpoint, it takes an independent checkpoint and does not communicate with other entities. This kind of checkpoint is what creates potential inconsistencies between latest checkpoints of tasks, requiring rollback to older checkpoints. When there is an interaction, then the destination entity of the dependency takes a checkpoint before actually allowing the interaction to affect the computation, if indeed the dependence is new. This is a heuristic to make the recovery line progress. Formally, if the interaction creates a dependency from entity i to entity j, carrying the sequence number x (from i), then entity j must take a checkpoint before completing the interaction causing the dependence provided that $ddv[j][i] \neq x$. This check detects that the independence

is new, it makes sure that not every dependency causing interaction leads to a forced checkpoint.

The recovery protocol is identical to that for uncoordinated checkpointing, described next.

4.3 Uncoordinated Checkpointing

In the case of uncoordinated checkpointing the checkpointing phase is straightforward and similar to the independent checkpointing phase of the communication induced checkpointing scheme.

In the recovery phase we need to recover to a globally consistent state. This means we need to compute a recovery line from the set of checkpoints stored. Indeed in this case a *domino effect* [10] may occur leading to a lot of lost computation. This is the potential overhead paid for no coordination at all in the checkpointing and communication phases. The recovery line can be computed in the traditional fashion by presenting the full set of timestamps to a central "decider process" and letting it compute the recovery line. The recovery line can be computed by constructing the checkpoint graph[4] from the timestamps. Then one can use reachability analysis to compute the set of checkpoints to recover from. Indeed if the entire set of timestamps is broadcast, then each entity can run this algorithm and decide for itself the point to which it should rollback. A typical uncoordinated checkpointing algorithm is optimized by implementing a garbage collection algorithm[4] that collects the timestamps and computes the recovery line. In the process it removes unnecessary history information. This too uses the same set of timestamps.

5 Implementation Notes

For efficiency reasons it is important to keep the overheads low in the checkpointing and recovery protocols, and during all interactions. In this section we evaluate the overheads introduced by the mechanisms explained in this paper. All interactions must carry a sequence number which is an integer. The code to be executed with the sequence number itself is also small as shown earlier. It is important to note that not all reads or writes need to incur this overhead. In a typical DSM implementation[8] based on sequential consistency with a

write-invalidate coherency protocol, the first write is associated with a page fault, leading to a *page grab*, invalidating other copies of the page. It is only at this time that we need to carry the sequence number. Successive writes can proceed without any overhead. Similarly a read request usually requires one to get a *read only* copy of the page. It is only during this *page get* that the overhead of sequence number management is required, not during successive reads.

Secondly, during checkpointing and recovery we need to store or share information in the form of the DDV. The size of the DDV therefore is a concern. Since tasks may interact with other tasks or pages, their DDV size will be as large as the number of tasks and the number of pages. For the pages, we need only maintain a DDV as large as the number of tasks, as pages only interact directly with tasks. This may represent a considerable savings for applications where the number of pages is large and the number of tasks is not very large. Recovery for coordinated checkpointing does not require exchanging of the DDV between entities. But uncoordinated checkpointing may require several DDV's from each entity to be sent to a central recovery manager for computing the recovery line. The alternate of distributed recovery will require several DDV's to be broadcasted from each entity. Hence keeping more smaller DDVs is beneficial. In fact one can keep DDV's in a compressed form (rather than explicit array representation) if the communication delays and latencies are high for the system.

6 Related work

In this section we compare the description in the current paper with other work in the area. Apart from the work of Gunaseelan and LeBlanc[5], this paper seems to be the only one that talks about timestamps for shared memory. Their paper deals explicitly with IPC based on RPC retaining shared information. They also talk about specific language support for their study. Also, they do not concern themselves with issues related to fault tolerance as such.

Janssens and Fuchs[7] talk about analyzing all the communication that takes place in a DSM system and abstracting it to the actual resulting inter task dependencies. Our approach is different in that we consider the pages

themselves as entities, like tasks, thereby making the basic building blocks from which the dependencies are constructed, different. We also note that they use timestamps with the aim of rebuilding the ownership information state of DSM pages. Both [7] and [6] talk only about coordinated checkpointing and recovery.

Our work borrows the ideas of timestamps and direct dependence vectors from the work of Baldoni *et. al.* [1]. Their aim was to compute the "*first global checkpoint that contains*" or "*follows*" a given set of local checkpoints. They also do not talk about application to DSM systems. Their work in turn borrows theoretical ideas from the work of Wang *et. al.* [11], in particular the fact that direct dependency information suffices for constructing consistent global checkpoints.

7 Conclusion

This paper describes common mechanisms for implementing a variety of checkpointing and rollback recovery protocols for both message passing and distributed shared memory systems. The idea is to treat pages as separate entities like tasks and provide a mechanism for tracking direct dependencies among tasks and memory pages. Apart from the fact that it is a mechanism common to both DSM and MP systems over a variety of protocols, it is efficient for implementation since the overhead for each interaction is very low - both in terms of the computation required and the amount of control information exchanged. We also note that the mechanism is sufficient to support several of the optimizations discussed in literature.

The mechanisms proposed in this paper are currently under implementation in the Gobelins cluster OS[8] which supports both DSM and message passing for IPC.

References

[1] R. Baldoni, G. Cioffi, J. Helary, and M. Raynal. Direct dependency-based determination of consistent global checkpoints. *International Journal of Computer Systems Sciences and Engineering*, 16(1):43–49, 2001.

[2] M. Banatre, A. Gefflaut, P. Joubert, C. Morin, and P.A. Lee. An architecture for tolerating processor failures in shared-memory multiprocessors. *IEEE Transaction on Computers*, 45(10):1101–1115, 1996.

[3] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Computer Systems*, 3(1):63–75, February 1985.

[4] M. Elnozahy, L. Alvisi, Y-M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-99-148, Carnegie Mellon University, June 1999.

[5] L. Gunaseelan and R.J. LeBlanc Jr. Event ordering in a shared memory distributed system. In *International Conference on Distributed Computing Systems*, pages 256–263, 1993.

[6] G. Janakiraman and Y. Tamir. Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. In *Symposium on Reliable Distributed Systems*, pages 42–51, 1994.

[7] B. Janssens and W.K. Fuchs. Reducing interprocessor dependence in recoverable distributed shared memory. In *Symposium on Reliable Distributed Systems*, pages 34–41, 1994.

[8] R. Lattiaux, C. Morin, and G. Vallée. Containers: An architecture for an efficient cluster operating system. Technical Report 1442, IRISA, February 2002.

[9] C. Morin and I. Puaut. A survey of recoverable distributed shared virtual memory systems. *IEEE Transaction on Parallel and Distributed Systems*, 8(9):959–969, 1997.

[10] B. Randell. System structure for software fault tolerance. *IEEE Trans. on Software Engineering*, 1(2):220–232, 1975.

[11] Y.M. Wang, A. Lowry, and W.K. Fuchs. Consistent global checkpoints based on direct dependency tracking. *Information Processing Letters*, 50:223–230, 1994.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399